# Less is More: Unparser-completeness of Metalanguages for Template Engines

B.J. Arnoldus M.G.J. van den Brand A. Serebrenik

Technische Universiteit Eindhoven
Eindhoven, The Netherlands
B.J.Arnoldus@alumnus.tue.nl, {M.G.J.v.d.Brand, A.Serebrenik}@tue.nl

## Abstract

A code generator is a program translating an input model into code. In this paper we focus on template-based code generators in the context of the model view controller architecture (MVC).

The language in which the code generator is written is known as a metalanguage in the code generation parlance. The metalanguage should be, on the one side, expressive enough to be of practical value, and, on the other side, restricted enough to enforce the separation between the view and the model, according to the MVC.

In this paper we advocate the notion of *unparser-complete metalanguages* as providing the right level of expressivity. An unparser-complete metalanguage is capable of expressing an unparser, a code generator that translates any legal abstract syntax tree into an equivalent sentence of the corresponding context-free language. A metalanguage not able to express an unparser will fail to produce all sentences belonging to the corresponding context-free language. A metalanguage able to express more than an unparser will also be able to implement code violating the model/view separation.

We further show that a metalanguage with the power of a linear deterministic tree-to-string transducer is unparser-complete. Moreover, this metalanguage has been successfully applied in a non-trivial case study where an existing code generator is refactored using templates.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—Code generation; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Pretty printers

***General Terms*** Languages

***Keywords*** code generation, templates, unparser

## 1. Introduction

Code generators are programs translating input models to code. They can be seen as meta-programs since they manipulate code, i.e. code occurs both as data (object code) and as an executed artifact (meta-code). One traditionally distinguishes between *homogeneous* and *heterogeneous* code generators [24]. In homogeneous code generators the metalanguage and object language coincide. In heterogeneous code generators the metalanguage and the object language may differ. A heterogeneous code generator can be implemented by means of, e.g., print statements, abstract syntax trees instantiation, term rewriting and templates [18, 21, 28]. In this paper we consider heterogeneous code generators, and more specifically, template-based code generators.

```
<ul>
#foreach( $product in $allProducts )
    <li>$product</li>
#end
</ul>
```

**Figure 1.** HTML code for a list of products.

One of the most well-known applications of template-based code generation is the HTML generation in dynamic web applications, where each HTML page is generated on a user request. Templates can be also used in broader context, e.g., for generating data model classes and state machine pattern implementations. In general, templates can be seen as fragments of object code intermitted with holes, containing instructions expressed in the metalanguage. When a *template engine* processes a template it directly emits the object code to the output, and evaluates the instructions in the holes with respect to the input data. For example, the template in Figure 1, expressed in the Velocity Template Language [25], generates HTML code for a list of products given in the input data. Numerous template-based engines are available, including JSP [4], Velocity [25] and StringTemplate [21]. Every template engine has its own metalanguage and evaluation strategy. With the notable exception of the metalanguage of StringTemplates, most metalanguages lack a formal requirements definition.

The main contribution of this paper consists in formalizing the requirements for a metalanguage of a template engine in the context of the model view controller architecture. Intuitively, the metalanguage should be, on the one side, expressive enough to be useful for practical applications, and, on the other side, it should be restricted enough, to disallow, e.g., the view to modify the model. We claim that the right balance can be found by demanding from the metalanguage to be *unparser-complete*, i.e., to be able to express an *unparser* and not more than an unparser. Section 2 presents the intuition behind this claim. After introducing a number of preliminary notions in Section 3, in Section 4 we formalize the notion of an unparser and in Section 5 the related notion of the *desugar* function. Unparser-completeness is introduced in Section 6 closing the discussion of the theoretical framework.

A complementary task of showing practical power of unparser complete metalanguages, is the second contribution of this paper. We have designed our own metalanguage discussed in Section 7 and conducted a number of case studies. One of these case studies,



*GPCE'11,* October 22–23, 2011, Portland, Oregon, USA.

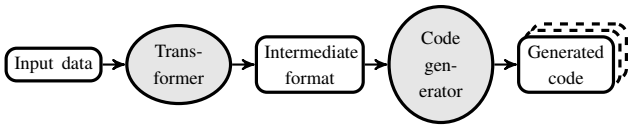


**Figure 2.** Two-stage architecture.

reimplementation of a finite-state machine generator `NunniFSMGen` is discussed in Section 8. Finally, related work is briefly reviewed in Section 9, and Section 10 concludes.

## 2. Requirements

In this section, we formulate the requirements for a template metalanguage. We focus on the usage of templates in the context of the model view controller architecture (MVC) [19] and heterogeneous code generators. In MVC architecture, templates are commonly used to implement the *view* of the internal data of an application (*model*). A view can be rendering the HTML for a set of data from the database in case of web applications (cf. the code fragment in Section 1), or converting a given abstract syntax tree into code.

The MVC architecture decouples the models and its transformations from the view components, hereby reducing the complexity and increasing the flexibility of the system. This separation of concerns also allows different views for the same underlying model. We show in Section 8 the benefits of MVC in the context of a code generator. The same model is used for different target languages, resulting in less code-clones than the original implementation.

While in the original paper on MVC [19] the *view* was expected to send editing messages to the model, already in [7] this functionality was restricted to the *controller*, and the *view* was only allowed to receive the messages from the model to update the way the model is shown. This intuition was formalized in [21], where it has been argued that the view should neither alter the model nor perform calculations depending on the semantics of the model. Unfortunately, the separation of view and logic is not enforced in existing template engines, such as JSP, i.e., it is possible to write JSP templates with all logic in a single file. Typically, such a file will contain fragments in HTML, JSP-tags, Java, and SQL. Not only does this file violate the MVC architecture principle, because logic (model) and presentation (view) are not separated, but it is also hard to understand the file due to different escaping characters required to support multiple programming languages, executed at different stages.

Separation of model and view results in a two-stage architecture (see Figure 2). In a two-stage architecture, the template is solely responsible for instantiating a view based on the intermediate representation. i.e. the model. The input model is provided by the first stage that can implement calculations and transformations. This separation is not formalized in this article, however, intuitively its place is at the point where the code generator only has to handle target language specific issues. For example, in case of a web application the first stage implements the business logic responsible for fetching data from a database and preparing it for the actual HTML and/or PDF generation, while the HTML generation is carried out during the second stage and PDF generation using another implementation in the second stage. It is also possible to have an n-stage architecture, although, the n-stage architecture can be seen as a two-stage architecture, where the transformer and/or the code generator exist of multiple stages.

In order to enforce the two-stage architecture, the language of the templates should be restricted. For instance, it should not include constructs for performing calculations, in order to make sure that all calculations are carried out before the templates are evaluated, i.e., during the first stage. However, in order to be practical, the language of the templates should be expressive enough. Indeed, a too restrictive metalanguage limits the applicability of the metalanguage, and thus of the template engine. Hence, we require the metalanguage to be able to implement code generators, which can produce *every* (correct) sentence of the output language. A code generator that can produce every sentence of the output language given an abstract syntax tree is called an *unparser*. The unparser is a special kind of code generator, transforming an abstract syntax tree to a concrete syntax, such that if the emitted concrete syntax is parsed then the input abstract syntax tree is obtained again, i.e. the semantics are not altered by the unparser. Unparsers are closely related to *pretty printers* [20] that in addition to producing every sentence of the output language given an abstract syntax tree take care of an appropriate layout.

## 3. Preliminaries

We start with a number of basic definitions and declarations of symbols used throughout this paper. We assume the reader to be familiar with the formal language theory and basics of the tree language theory [9, 15]. In this section we present the notation used and recall the most important definitions.

Integer variables are denoted $k, i, j, p$ and $r$. Formal languages are sets of words, i.e., finite sequences of symbols derived from a finite set of symbols, known as an *alphabet* and denoted as $\Sigma$. The set of all words over $\Sigma$ is denoted $\Sigma^*$. Every symbol $c$ of $\Sigma$ is associated with a unique non-negative integer, known as the *rank* of $c$ and denoted $r_c$. The rank is equal to the number of children a node representing the symbol will have. We further use $f$ to denote alphabet symbols with rank greater than 0. $\Sigma_r$ for the set of symbols of rank $r$. $X$ is a set of symbols called variables and we assume that the sets $X$ and $\Sigma_0$ are disjoint. $x$ is a variable $x \in X$ and is not used for integer values.

An important class of formal languages are the *context-free languages*, i.e., languages specified by a context-free grammar [15]:

**Definition 3.1** (Context-free grammar and language)**.** A context-free grammar (CFG) is a four-tuple $\langle \Sigma, N, S, \mathit{Prods} \rangle$, where $\Sigma$ is the alphabet, $N$ is a finite set such that $N \cap \Sigma = \emptyset$, $S \in N$ is the start symbol and *Prods* is a finite set of production rules of the form $n \to z$ where $n \in N$ and $z \in (N \cup \Sigma)^*$. A context-free language $\mathcal{L}(G)$ is the set of words generated by the context-free grammar $G$. The set $N$ is known as the set of *nonterminal symbols*.

For the sake of simplicity in this paper we focus on LALR grammars [10]. Our approach can be adapted to grammars with ambiguity by introducing ambiguity nodes and indicating the preferred derivations similarly to [26].

A *regular tree language* is a set of trees generated by a *regular tree grammar* [9]:

**Definition 3.2.** (Regular tree grammar). A regular tree grammar (RTG) is a four-tuple $\langle \Sigma, N, S, \mathit{Prods} \rangle$, where $\Sigma$ is the alphabet; $N$ is a finite set of nonterminal symbols with rank $r = 0$ and $N \cap \Sigma = \emptyset$; $S \in N$ is a start symbol; *Prods* is a finite set of production rules of the form $n \to t$, where $n \in N$ and $t \in \mathit{Tr}(\Sigma \cup N)$, where $\mathit{Tr}(\Sigma \cup N)$ is the set of trees over $\Sigma$ and $N$.

**Example 3.3.** Let $G$ be $\langle \Sigma, N, S, \mathit{Prods} \rangle$, where $\Sigma = \{a, b\}$ with $r_a = 0$ and $r_b = 2$, $N = \{S\}$ and $\mathit{Prods} = \{S \to b(S, S), S \to a\}$. The grammar $G$ is a regular tree grammar representing binary trees, and its language

$$\mathcal{L}(G) = \{a, b(a, a), b(b(a, a), a), b(a, b(a, a)), \ldots\}$$

is a regular tree language. ■

Next we introduce the notion of a linear tree homomorphism:

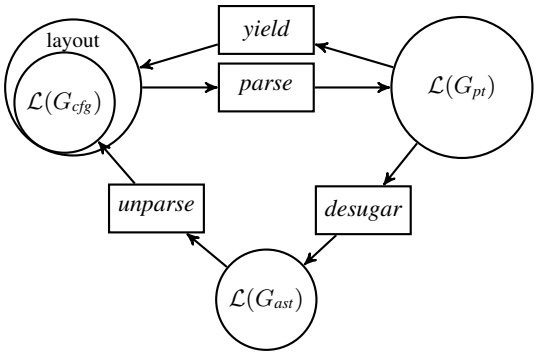


**Figure 3.** Relations between languages and their grammars.

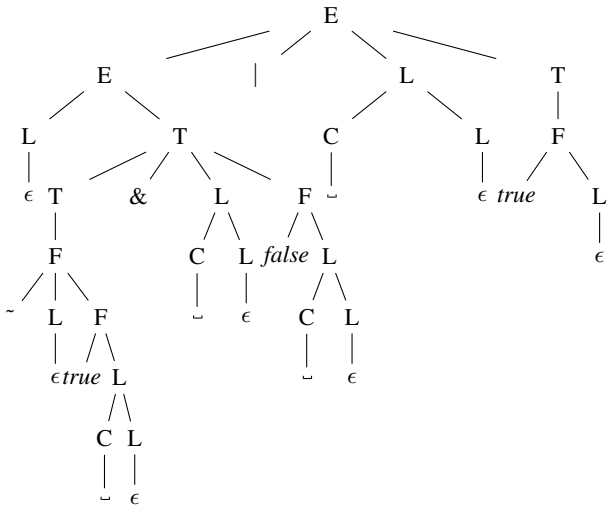


**Figure 4.** A parse tree of `~true & false | true` obtained with respect to the production rules of Example 4.1.

**Definition 3.4.** (Tree homomorphism) [9]. Let $\Sigma$ and $\Sigma'$ be two not necessarily disjoint ranked alphabets. For each $k > 0$ such that $\Sigma$ contains a symbol of rank $k$, we define a set of variables $X_k = \{x_1, \ldots, x_k\}$ disjoint from $\Sigma$ and $\Sigma'$.

Let $h_\Sigma$ be a mapping which, with $c \in \Sigma$ of rank $k$, associates a term $t_c \in \mathit{Tr}(\Sigma', X_k)$. The tree homomorphism $h : \mathit{Tr}(\Sigma) \to \mathit{Tr}(\Sigma')$ is determined by $h_\Sigma$ as follows:

- $h(a) = t_a \in \mathit{Tr}(\Sigma')$ for each $a \in \Sigma$ of rank 0,
- $h(c(t_1, \ldots, t_n)) = t_c\{x_1 \leftarrow h(t_1), \ldots, x_k \leftarrow h(t_k)\}$ where $t_c\{x_1 \leftarrow h(t_1), \ldots, x_k \leftarrow h(t_k)\}$ is the result of applying the substitution $\{x_1 \leftarrow h(t_1), \ldots, x_k \leftarrow h(t_k)\}$ to the term $t_c$.

$h_\Sigma$ is called a *linear tree homomorphism* when no $t_c$ contains two occurrences of the same $x_k$. Thus a linear tree homomorphism cannot duplicate trees.

**Example 3.5.** (Tree homomorphism) [9]. Let $\Sigma = \{g(,,), a, b\}$ and $\Sigma' = \{f(,), a, b\}$. Let us consider the tree homomorphism $h$ determined by $h_\Sigma$ defined by: $h_\Sigma(g) = f(x_1, f(x_2, x_3)), h_\Sigma(a) = a, h_\Sigma(b) = b$. For instance, we have: If $t = g(a, g(b, b, b), a)$, then $h(t) = f(a, f(f(b, f(b, b)), a))$.

## 4. Unparser

As already stated in Section 2, our expressivity requirements for metalanguages of template engines are based on the notion of the *unparser*. An unparser translates an abstract syntax tree representing a sentence into a textual representation of the sentence [6]. An unparser can be seen as a part of the "circle of code" shown in Figure 3, i.e. the circle from concrete syntax to parse tree to abstract syntax and back to concrete syntax. The textual representation of a program obeying a context-free grammar ($\mathcal{L}(G_{cfg})$ in Figure 3) can be *parse*d to obtain the corresponding parse tree ($\mathcal{L}(G_{pt})$). The set of parse trees of a context-free grammar is a regular tree language [9].

Since the parse tree contains exactly the same information as the textual representation, the textual representation can be restored using the parse tree. The corresponding mapping from $\mathcal{L}(G_{pt})$ to $\mathcal{L}(G_{cfg})$ is called *yield* [9]. Alternatively, one can *desugar* the parse tree, i.e., simplify it by removing the semantically-irrelevant layout information. In this way, an abstract syntax tree ($\mathcal{L}(G_{ast})$) is obtained. Finally, the *unparse* function closes the circle and maps elements of $\mathcal{L}(G_{ast})$ to $\mathcal{L}(G_{cfg})$.

**Example 4.1.** To illustrate the functions in Figure 3 consider the following set of production rules:

| | | |
|---|---|---|
| E → L T | F → "~" L F | L → C L |
| E → E "\|" L T | F → "(" L E ")" L | L → ϵ |
| T → F | F → "*true*" L | C → "␣" |
| T → T "&" L F | F → "*false*" L | C → "\n" |

The parse tree of `~true & false | true` with respect to these production rules is shown in Figure 4. In the running text we also write a parse tree as a term $n(s_1, t'_1, s_2, \ldots, s_r, t'_r, s_{r+1})$, where $t'_1 \ldots t'_r$ are sub parse trees with top nonterminals $n_1 \ldots n_r$ and strings $s_1 \ldots s_{r+1}$ are the terminals.

The sentence `~true & false | true` can be restored by applying *yield* to the tree in Figure 4. The *desugar* function provides a mapping of concrete syntax constructs (e.g., & and *true*) to abstract syntax constructs (*And* and *True*, respectively), applying the *desugar* function to the parse tree results in the abstract syntax tree *Or*(*And*(*Not*(*True*), *False*), *True*). The abstract syntax tree can be unparsed to `~true&false|true`. Observe that while the application of *unparse*∘*desugar*∘*parse* produces a semantically equivalent sentence, layout preservation is not guaranteed. ■

As suggested by the example, an abstract syntax tree corresponds to multiple semantically-equivalent textual representations, while *unparse* necessarily produces only one of these representations. This is why the set of context-free sentences is larger than the range of *unparse*, i.e.,

$$\mathcal{L}(G_{cfg}) \supseteq \mathit{unparse}(\mathit{desugar}(\mathit{parse}(\mathcal{L}(G_{cfg})))) \quad (1)$$

Moreover, the unparser is correct if and only if re-parsing its output sentences produce the same abstract syntax trees as the original inputs [22]. This correctness requirement means that the combination of *unparse* and *desugar* should not alter the meaning of the code represented by the concrete syntax or the abstract syntax:

$$\mathcal{L}(G_{ast}) = \mathit{desugar}(\mathit{parse}(\mathit{unparse}(\mathcal{L}(G_{ast})))) \quad (2)$$

Our intention is to derive the *unparse* function from the context-free grammar of the output language. However, in order to generate a textual representation from an abstract syntax tree the unparser should be aware of the mapping between concrete syntax constructs and abstract syntax constructs, akin to the one mentioned in Example 4.1. One possible way to construct the mapping is by means of heuristics [29]. However, this approach will introduce machine generated names for the abstract syntax tree nodes. In order to retain the full control of the abstract syntax constructs, we propose to

integrate abstract syntax constructs (signature labels) in the production rules of a context-free grammar. Formally, a production rule in the augmented context-free grammar has the form $n \rightarrow z\{c\}$ where $n \in N$, $z \in (N \bigcup \Sigma)^*$ and $c$ is an element of an alphabet $\Sigma'$. The set $\Sigma'$ is the alphabet of the regular tree grammar belonging to the abstract syntax and is not necessarily disjoint from $N \bigcup \Sigma$. We furthermore require a signature label $c$ to be used at most once. In order to remove the layout syntax and other superfluous syntax, it is allowed, under strict conditions, to have production rules without signature labels. These conditions are in the case of:

- ◇ **Layout syntax** - The nonterminals belonging to the layout syntax, such as whitespaces and comment, should not be defined with a signature label. The desugar function, as defined in Section 5, removes these syntax from the tree. It is mandatory, that the layout syntax includes the empty string $\epsilon$, as the layout is not restored by the automatic derived unparser.
- ◇ **Chain rules** - It is allowed that production rules of the form $n_1 \rightarrow n_2$, are not accompanied with a signature label. The abstract syntax belonging to $n_2$ is propagated to $n_1$, which is allowed, since all signature labels are unique. Furthermore, it is allowed that $n_2$ is surrounded by layout nonterminals, as these layout syntaxes contain the empty string.

**Example 4.2.** (Augmented context-free grammar)

| | | |
|---|---|---|
| E → L T | F → "~" L F {*Not*} | L → C L |
| E → E "\|" L T {*Or*} | F → "(" L E ")" L {*Br*} | L → $\epsilon$ |
| T → F | F → "*true*" L {*True*} | C → "␣" |
| T → T "&" L F {*And*} | F → "*false*" L {*False*} | C → "\n" |

The production rules above show the extension of a context-free grammar of Example 4.1 with signature labels. ■

In presence of signature labels we further adapt the term notation for the parse trees and write $parse(s_1 \cdot s'_1 \cdot s_2 \cdot \ldots \cdot s_r \cdot s'_r \cdot s_{r+1}) =< n, c > (s_1, t'_1, s_2, \ldots, s_r, t'_r, s_{r+1})$, where $n$ is the top non-terminal, $t'_1 \ldots t'_r$ are sub parse trees with top nonterminals $n_1 \ldots n_r$, strings $s_1 \ldots s_{r+1}$ are the terminals and $c$ is the label associated with $n \rightarrow s_1, n_1, s_2, \ldots, s_r, n_r, s_{r+1}$. If there is no such label, $parse(s_1 \cdot s'_1 \cdot s_2 \cdot \ldots \cdot s_r \cdot s'_r \cdot s_{r+1})$ is defined as $n(s_1, t'_1, s_2, \ldots, s_r, t'_r, s_{r+1})$. Furthermore, the function *parse* used in this article is complete.

For a context-free grammar extended with signatures we can derive *unparse*. For each production rule in the context-free grammar there is a case in the definition of *unparse*, called an *action*, allowing *unparse* to traverse the abstract syntax tree and to restore terminals whenever needed. For instance, in Example 4.1, given the production rule T → T "&" L F {*And*}, the definition of the unparser should have an action for $unparse(And(x, y))$. In general, each action in *unparse* has a left-hand side and a right-hand side, see Example 4.5. First, we provide the definition to derive an unparser from a given context-free grammar.

**Definition 4.3.** (Unparse). Let $G_{cfg} = \langle \Sigma, N, S, Prods \rangle$ be a context-free grammar augmented with signature labels, where $N'$ is the set of non-terminals defined by the production rules with a signature label. Then, the corresponding function *unparse* is defined by a set of actions *Actions* such that for any $n \rightarrow z_1 \ldots z_k\{c\} \in Prods$

- ◇ either $mkLhs(z_1, \ldots, z_k, 1) \neq \epsilon$ and $unparse(c(mkLhs(z_1, \ldots, z_k, 1))) = mkRhs(z_1, \ldots, z_k, 1) \in Actions$,
- ◇ or $mkLhs(z_1, \ldots, z_k, 1) = \epsilon$ and $unparse(c) = mkRhs(z_1, \ldots, z_k, 1) \in Actions$,

where

$$mkLhs(i) = \epsilon$$
$$mkLhs(z_1, z_2, \ldots, z_j, i) = \begin{cases} mkLhs(z_2, \ldots, z_j, i+1) & \text{if } z_1 \notin N' \\ x_i, mkLhs(z_2, \ldots, z_j, i+1) & \text{if } z_1 \in N' \text{and } mkLhs(z_2, \ldots, z_j, i+1) \neq \epsilon \\ x_i & \text{otherwise} \end{cases}$$

and

$$mkRhs(i) = \epsilon$$
$$mkRhs(z_1, z_2 \ldots, z_j, i) = \begin{cases} z_1 \cdot mkRhs(z_2, \ldots, z_j, i+1) & \text{if } z_1 \in \Sigma \\ unparse(x_i) \cdot mkRhs(z_2, \ldots, z_j, i+1) & \text{if } z_1 \in N' \\ mkRhs(z_2, \ldots, z_j, i+1) & \text{if } z_1 \notin (N' \cup \Sigma) \end{cases}$$

and $\cdot$ denotes the string concatenation operation.

**Example 4.4.** We illustrate Definition 4.3 with the production rule T → T "&" L F {*And*} from Example 4.2. Then, the set $N'$ of non-terminals corresponding to production rules with signature information is {E, F, T}. Hence, "&" and L should be omitted from the left-hand side of the unparser action:

$$\begin{aligned} mkLhs(\text{T}, \text{“\&”}, \text{L}, \text{F}, 1) &= x_1, mkLhs(\text{“\&”}, \text{L}, \text{F}, 2) \\ &= x_1, mkLhs(\text{L}, \text{F}, 3) = \\ &= x_1, mkLhs(\text{F}, 4) = \\ &= x_1, x_4 \end{aligned}$$

since $mkLhs(5) = \epsilon$. Similarly, for the right-hand side of the action we ignore L:

$$\begin{aligned} mkRhs(\text{T}, \text{“\&”}, \text{L}, \text{F}, 1) &= unparse(x_1) \cdot mkRhs(\text{“\&”}, \text{L}, \text{F}, 2) = \\ & unparse(x_1) \cdot \text{“\&”} \cdot mkRhs(\text{L}, \text{F}, 3) = \\ & unparse(x_1) \cdot \text{“\&”} \cdot mkRhs(\text{F}, 4) = \\ & unparse(x_1) \cdot \text{“\&”} \cdot unparse(x_4) \cdot mkRhs(5) = \\ & unparse(x_1) \cdot \text{“\&”} \cdot unparse(x_4) \cdot \epsilon = \\ & unparse(x_1) \cdot \text{“\&”} \cdot unparse(x_4) \end{aligned}$$

Since $mkLhs(\text{T}, \text{“\&”}, \text{L}, \text{F}, 1) \neq \epsilon$, the action corresponding to T → T "&" L F {*And*} is $unparse(And(x_1, x_4)) = unparse(x_1) \cdot \text{“\&”} \cdot unparse(x_4)$. ■

In general, the *left-hand side* matches on a node in the abstract syntax tree with the signature label $c$ and the variables $x_1, \ldots, x_k$ are assigned to the subtrees belonging to the label $c$. Following the previous definition, the rank of $c$, i.e., the number of arguments of $c$ created by *mkLhs*, is not necessarily equal to $k$: the rank of $c$ is equal to the number of nonterminals in the pattern of the production rule labeled $c$ in the augmented context-free grammar. Therefore, the variable $x_i$ only exists in the action $unparse(c(\ldots)) = \ldots$ if a symbol at index $i$ in the right-hand side of the corresponding production rule is a nonterminal $n \in N'$. The *right-hand side* constructs a string $s_1 \cdot \ldots \cdot s_k$. The number of strings $k$ is equal to the number of symbols in the pattern of the corresponding production rule. Each $s_i$ is either a string or a recursive unparser invocation. Specifically, $s_i$ is a string, if a terminal is defined at index $i$ in the production rule; an unparser invocation, if a nonterminal $n \in N'$ is defined at position $i$, and $\epsilon$ for the remaining case if $n_i \notin (N' \cup \Sigma)$.

**Example 4.5.** Example 4.4, continued. Given the aforementioned production rules, the following unparser is derived:

$$\begin{aligned} unparse(Not(x_3)) &= \text{“~”} \cdot unparse(x_3) \\ unparse(And(x_1, x_4)) &= unparse(x_1) \cdot \text{“\&”} \cdot unparse(x_4) \\ unparse(Or(x_1, x_4)) &= unparse(x_1) \cdot \text{“\|”} \cdot unparse(x_4) \\ unparse(Br(x_3)) &= \text{“(”} \cdot unparse(x_3) \cdot \text{“)”} \\ unparse(True) &= \text{“true”} \\ unparse(False) &= \text{“false”} \end{aligned}$$

■

The unparser derived according to Definition 4.3 is linear and deterministic. We say that an unparser is *linear* if for each action in the unparser and every $x_i$ in the action, $x_i$ occurs not more than once in the action's right-hand side. The unparser is called *deterministic* if actions have incompatible left-hand sides, i.e., for every tree there exists only one applicable unparser action.

**Theorem 4.6.** The unparser derived according to Definition 4.3 is linear and deterministic.

*Proof.* Linearity follows from Definition 4.3: every variable appearing on the right hand side appears only once. Recall that a signature label $c$ is used once in the augmented context-free tree grammar. A signature label $c$ directly corresponds to one action in $\mathit{unparse}(c(x_1, \ldots, x_k))$. Since a signature label $c$ is used for only one production rule, the left-hand sides of the unparser are unique and thus the unparser is deterministic. □

Definition 4.3 also ensures that *unparse* always terminates if its argument is a finite tree. Recall that $\mathit{unparse}(\ldots)$ for $n \rightarrow z_1 \ldots z_k\{c\}$ distinguishes between $\mathit{mkLhs}(z_1, \ldots, z_k, 1) = \epsilon$ and $\mathit{mkLhs}(z_1, \ldots, z_k, 1) \neq \epsilon$. However, $\mathit{mkLhs}(z_1, \ldots, z_k, 1) = \epsilon$ holds only if $z_i \in \Sigma$ for all $i$, $1 \leq i \leq k$. Therefore, the right-hand side expression $\mathit{mkRhs}(z_1, z_2 \ldots, z_k, 1) = z_1 \cdot \ldots \cdot z_k$ and *unparse* is defined as $\mathit{unparse}(c) = z_1 \cdot \ldots \cdot z_k$. Since the right-hand side of the latter equation does not contain calls to *unparse*, it cannot introduce non-termination. The remaining case we have to consider is $\mathit{mkLhs}(z_1, \ldots, z_k, 1) \neq \epsilon$. In this case *unparse* is defined as $\mathit{unparse}(c(\mathit{mkLhs}(z_1, \ldots, z_k, 1))) = \mathit{mkRhs}(z_1, \ldots, z_k, 1)$. Termination stems from the fact that every variable appearing on the right hand-side appears in the left-hand side, and from the reduction in the term size between the left-hand side and the right-hand side terms.

We cannot yet prove that the unparser of Definition 4.3 satisfies (1) and (2) as the *desugar* function is still to be defined.

## 5. Desugaring

The *desugar* function can be manually defined in the parser definition, like in parser implementations such as YACC [17]. These parsers allow to associate a *semantic action* with a production rule in the grammar, such that the semantic actions can directly instantiate an abstract syntax tree. A manually defined *desugar* function must define a linear tree homomorphism, as we argue later on, otherwise regularity of the abstract syntax tree is not guaranteed.

Having a context-free grammar with signature labels, the abstract syntax tree can be automatically instantiated from a parse tree. This is executed by the *desugar* function, which replaces the nodes in the parse tree with new nodes, which are labeled by signature labels. The rank of signature label $c$ is equal to the number of nonterminals in the corresponding production rule of the context-free grammar. Nodes in the parse tree without a signature label are removed from the tree. This mechanism is responsible for removing the nodes that do not contain semantically significant information, such as chain rules and layout syntax.

**Definition 5.1.** (Desugar). The *desugar* function is defined by the following equations:

$$
\begin{array}{l}
\mathit{desugar}(x) = \epsilon \qquad \text{if } x \in \Sigma \\
\mathit{desugar}(f(x_1, \ldots, x_k)) = \mathit{dc}(x_1, \ldots, x_k) \\
\mathit{desugar}(< f, c > (x_1, \ldots, x_k)) = \\
\quad \begin{cases} c & \text{if } \mathit{dc}(x_1, \ldots, x_k) = \epsilon \\ c(\mathit{dc}(x_1, \ldots, x_k)) & \text{if } \mathit{dc}(x_1, x_2, \ldots, x_k) \neq \epsilon \end{cases}
\end{array}
$$

and

$$
\begin{array}{l}
\mathit{dc}() = \epsilon \\
\mathit{dc}(x_1, x_2, \ldots, x_k) = \\
\quad \begin{cases} \mathit{dc}(x_2, \ldots, x_k) & \text{if } x_1 \in \Sigma \\ \mathit{desugar}(x_1), \mathit{dc}(x_2, \ldots, x_k) & \text{if } x_1 \notin \Sigma \text{ and } \mathit{dc}(x_2, \ldots, x_k) \neq \epsilon \\ \mathit{desugar}(x_1) & \text{otherwise} \end{cases}
\end{array}
$$

**Example 5.2.** (Desugar). Applying the *desugar* function to the parse tree $t$ of `~true & false | true` using the grammar of Example 4.2 will result in the abstract syntax tree: $\mathit{desugar}(t) = \mathit{Or}(\mathit{And}(\mathit{Not}(\mathit{True}), \mathit{False}), \mathit{True})$. ■

Observe that Definition 5.1 ensures termination of *desugar* as long as its argument is a finite tree. Indeed, each subsequent call to *desugar* or *dc* reduces the size of the input argument either by removing the function symbol, e.g., $\mathit{desugar}(f(x_1, \ldots, x_k)) = \mathit{dc}(x_1, \ldots, x_k)$ or by reducing the number of arguments in the call, e.g., $\mathit{dc}(x_1, x_2, \ldots, x_k) = \mathit{dc}(x_2, \ldots, x_k)$.

**Theorem 5.3.** The abstract syntax tree obtained by applying the *desugar* function to a parse tree belongs to a regular tree language [8].

*Proof.* Recognizability of trees by finite tree automata is closed under linear tree homomorphism [11]. The *desugar* function is a linear tree homomorphism; subtrees are only removed and not duplicated. Since the abstract syntax tree is a linear tree homomorphism of the parse tree and the set of parse trees of a context-free language is a regular tree language [9], the abstract syntax tree belongs to a regular tree language. □

## 6. Unparser Completeness

We call a metalanguage capable to express unparsers *unparser-complete*. In this section we show that unparser-completeness is a *more* restricted notion than Turing-completeness. To establish this result we (1) show that the unparser as defined in Definition 4.3 can be expressed by a linear deterministic top-down tree-to-string transducer, and (2) recall that the top-down tree-to-string transducer is strictly less powerful than a Turing machine, i.e., top-down-tree-to-string transducers accept are a subset of the languages Turing machines can accept [27].

**Definition 6.1.** (Top-down tree-to-string transducer) [12]. A top-down tree-to-string transducer is a 5-tuple $M = \langle Q, \Sigma, \Sigma', q_0, R \rangle$, where $Q$ is a finite set of states, $\Sigma$ is the ranked input alphabet, $\Sigma'$ is the output alphabet, $q_0 \in Q$ is the initial state, and $R$ is a finite set of rules of the form:

$$q(\sigma(x_1, \ldots, x_k)) \rightarrow s_1 q_1(x_{i_1}) s_2 q_2(x_{i_2}) \ldots s_p q_p(x_{i_p}) s_{p+1}$$

with $k, p \geq 0$; $q, q_1, \ldots, q_p \in Q$; $\sigma \in \Sigma_k$; $s_1, \ldots, s_{p+1} \in \Sigma'^*$, and $1 \leq i_j \leq k$ for $1 \leq j \leq p$ (if $k = 0$ then the left-hand side is $q(c)$). $M$ is called *deterministic* if different rules in $R$ have different left-hand sides. $M$ is called *linear* if, for each rule in $R$, no $x_i$ occurs more than once in its right-hand side.

**Example 6.2.** Unparser in Example 4.5 can be seen as a top-down tree-to-string transducer $\langle Q, \Sigma, \Sigma', q_0, R \rangle$ such that the set of states $Q = \{\mathit{unparse}\}$, the input alphabet $\Sigma = \{\mathit{Not}, \mathit{And}, \mathit{Or}, \mathit{Br}, \mathit{True}, \mathit{False}\}$, the output alphabet $\Sigma' = \{$"~", "&", "|", "(", ")", "true", "false"$\}$ and the finite set of rules $R$ is given by actions defining the unparser in Example 4.5.

Next we show that for each context-free grammar an unparser can be defined using a linear and deterministic top-down tree-to-string transducer. Furthermore, any unparser corresponding to Definition 4.3 can be mapped on a top-down tree-to-string transducer.

**Theorem 6.3.** An unparser based on a linear deterministic top-down tree-to-string transducer can be defined for every context-free grammar augmented with signature labels.

*Proof.* We show that $\mathcal{L}(G_{ast}) = desugar(parse(unparse(\mathcal{L}(G_{ast}))))$ for the production rules of the form $n \rightarrow s_1 n_1 s_2 \ldots s_r n_r s_{r+1}\{c\}$.

Every production rule in a context-free grammar can be projected on the form $n \rightarrow s_1 n_1 s_2 \ldots s_r n_r s_{r+1}\{c\}$, where $s_1, \ldots, s_{r+1}$ are strings and may be the empty string $\epsilon$, and $n, n_1, \ldots, n_r$ are the nonterminals. In case the pattern $s_1 s_2$ occurs, the strings can be concatenated into a new string $s'_1$. We assume that the augmented grammar meets the requirements for augmenting a grammar with signature labels as sketched in Section 4. The abstract syntax tree belonging to this production rule $t_{ast} = c(t_1, \ldots, t_r)$, where $t_1, \ldots, t_r$ are the abstract syntax trees belonging to $n_1 \ldots n_r$. The corresponding tree-to-string transducer rule is: $q(c(x_1, \ldots, x_r)) \rightarrow s_1 q_1(x_1) s_2 \ldots s_r q_r(x_r) s_{r+1}$, where $q, q_1, \ldots, q_r$ are transducer states. Application of the transducer to the abstract syntax tree consists in *matching* the tree against the pattern $c(x_1, \ldots, x_r)$ and *replacing* it with a string originating from $s_1 q_1(x_1) s_2 \ldots s_r q_r(x_r) s_{r+1}$, where $q_1(x_1), \ldots, q_r(x_r)$ have been recursively applied to $t_1, \ldots, t_r$, i.e., $q(c(t_1, \ldots, t_r)) = s_1 \cdot s'_1 \cdot s_2 \cdot \ldots \cdot s_r \cdot s'_r \cdot s_{r+1}$, where $s'_1 = q_1(t_1) \ldots s'_r = q_r(t_r)$. In Section 7 this match-replace intuition will be used to define an eponymous construct in our unparser-complete metalanguage.

Parsing $s_1 \cdot s'_1 \cdot s_2 \cdot \ldots \cdot s_r \cdot s'_r \cdot s_{r+1}$ produces a parse tree $parse(s_1 \cdot s'_1 \cdot s_2 \cdot \ldots \cdot s_r \cdot s'_r \cdot s_{r+1}) =< n, c > (s_1, t'_1, s_2, \ldots, s_r, t'_r, s_{r+1})$, where $t'_1 \ldots t'_r$ are sub parse trees with top nonterminals $n_1 \ldots n_r$ and strings $s_1 \ldots s_{r+1}$ are the terminals. The abstract syntax tree is $desugar(< n, c > (s_1, t'_1, s_2, \ldots, s_r, t'_r, s_{r+1})) = c(t_1 \ldots t_r)$, where $t_1 = desugar(t'_1)$, $\ldots$, $t_r = desugar(t'_r)$, which is equal to the original abstract syntax tree. Since this relation holds for every production rule in a context-free language, the unparser can be defined using a top-down tree-to-string transducer for every context-free language. □

The proof that $\mathcal{L}(G_{cfg}) \supseteq unparse(desugar(parse(\mathcal{L}(G_{cfg}))))$ also holds is very similar to the proof of Theorem 6.3. One should take the string $s$ as starting point instead of the abstract syntax tree. The superset relation is a result of the fact that layout is not available in the abstract syntax tree and as a result it cannot be literally restored during unparsing. The language produced by the unparser is thus always a sentence of $\mathcal{L}(G_{cfg})$, but the set of sentences of $\mathcal{L}(G_{cfg})$ is greater than the set of sentences the unparser can produce.

**Theorem 6.4.** The relation $\mathcal{L}(G_{cfg}) \supseteq unparse(desugar(parse(\mathcal{L}(G_{cfg}))))$ holds for the unparser.

*Proof.* First, similarly to the proof of Theorem 6.3, the relation

$$\mathcal{L}(G_{ast}) = desugar(parse(unparse(\mathcal{L}(G_{ast}))))$$

holds when using a context-free grammar without production rules for layout syntax. Next we extend $\mathcal{L}(G_{cfg})$ with layout syntax resulting in $\mathcal{L}(G_{cfg})'$, then $\mathcal{L}(G_{cfg})' \supset \mathcal{L}(G_{cfg})$, since every sentence without layout must be in $\mathcal{L}(G_{cfg})'$, otherwise the languages are not semantical equal. Thus every sentence the unparser produce must be at least in $\mathcal{L}(G_{cfg})$, otherwise the *unparse* function does not meet the requirement of the unparser to be semantically transparent. □

The last step is that we show that the unparser is linear and deterministic.

**Theorem 6.5.** The unparser of Definition 4.3 is a linear and deterministic top-down tree-to-string transducer.

```
<: match :>
  <: Matchpattern =:> String
  ...
  <: Matchpattern =:> String
<: end :>
```

**Figure 5.** Match-replace construct.

*Proof.* The derivation of an unparser using Definition 4.3 can be mapped on a top-down tree-to-string transducer. Considering Definition 4.3 the unparser contains actions of the form:

$$unparse(c) = s$$
$$unparse(c(x_1, \ldots, x_k)) = s_1 \cdot unparse(x_1) \cdot \ldots \cdot s_k \cdot unparse(x_k) \cdot s_{k+1}$$

The similarity with the top-down tree-to-string transducer is obvious. Substitute the occurrences of *unparse* by states named $q$ and the unparser becomes a tree-to-string transducer.

The unparser is linear, since each $x_i$ occurs once on the left-hand side and once on the right-hand side.

The unparser is also deterministic, since it is derived from a context-free grammar augmented with signature labels, where each signature label is only used for one production rule. □

These theorems show that an unparser can be specified using a linear deterministic top-down tree-to-string transducer.

So far we have shown that the unparser as defined in Definition 4.3 can be expressed by a linear deterministic top-down tree-to-string transducer. Recall that the top-down tree-to-string transducer is strictly less powerful than the Turing machine, i.e., top-down-tree-to-string transducers accept a subset of the languages Turing machines can accept [27]. Indeed, the class of tree languages a top-down tree-to-string transducer can recognize is equal to its corresponding finite tree automaton [12]. Unlike a Turing machine, a top-down tree-to-string transducer cannot change the input tree on which it operates but only emit a string while processing the input tree. The class of languages the top-down tree-to-string transducer accepts is the class of *path-closed tree languages* [27], being a subset of regular tree languages [9]. One can show that the languages of abstract syntax trees of the augmented grammar are path-closed, since a signature label is only used for one production rule.

## 7. Metalanguage

To verify the usefulness of unparser-completeness metalanguages in practice, we have designed a metalanguage for templates and applied it in a number of case studies, including a redesign of a domain-specific language for web information systems, reimplementation of the Java-back end of a tree-like data structures manipulation library ApiGen [5], dynamic XHTML generation and reimplementation of the state-machine-based code generator NunniFSMGen. In the current section we focus on the metalanguage itself, while in Section 8 we present the NunniFSMGen case study, and discussion of other case studies can be found in [2]

The metalanguage provides three constructs: match-replace, subtemplate invocation and substitution. The *match-replace* (Figure 5) is a construct containing a set of match-rules with a tree pattern and an accompanying result string. The result string may contain metalanguage constructs, which are evaluated recursively.

The match-replace matches the match-patterns against the current input tree in the context. At the start of evaluation, this is the complete input tree. In case a match-pattern is valid, the match-replace starts evaluating the string belonging to the match-rule. In the match-pattern variables may be bound, which can be used while

evaluating the string belonging to the match-rule. After the string is evaluated, the result is used to replace the match-replace. This construct enables tree-matching, like the left-hand side of the state rules of the tree-to-string transducer.

The second construct is *subtemplate invocation*. Subtemplates allows to divide a template in multiple smaller units, but more important for unparser-completeness, is that this construct enables recursion. Two constructs are necessary to implement subtemplates, namely the declaration of subtemplates and the subtemplate call. The concrete syntax of a subtemplate declaration is `IdCon[ String ]`, while the syntax of a subtemplate call is `<: IdCon( Expr ) :>`. For instance, Figure 6 contains a subtemplate declaration *unparse* [...] , Lines 1–10 and five recursive subtemplate calls, e.g., `<:`*unparse*(*$x*)`:>` in Line 3. The template evaluator selects the subtemplate based on the identifier and it replaces the subtemplate call with the string resulting from the evaluated subtemplate. The `Expr` is used to obtain a new input data context for evaluating the subtemplate. Subtemplates support meta-variable lookups, that return either a subtree of the input data or a terminal, i.e. a string value. For usability reasons, the `Expr` supports more operations than necessary. The constructs of declaring string constants and string concatenations (`||`) do not allow to change the input data tree, and thus not break unparser-completeness. Note that the template engine preserves layout information defined in the string and no additional layout, other than specified in the string, is emitted to the output.

The match-replace and subtemplate constructs are sufficient to implement unparsers, and, hence, any metalanguage implementing these constructs is (at least) unparser-complete. Figure 6 shows an unparser for the running example language expressed in the given template metalanguage. For the ease of comprehension, in Figure 6 and the subsequent figures demonstrating code fragments in the template metalanguage, we adhere to the following typesetting convention. Structural elements of the metalanguage, such as match and `<:` are typeset in the boldface; names of subtemplates and variables are typeset in the italics; labels are underlined, and, finally, the terminals, i.e., elements being directly placed in the output are typeset in roman.

For usability reasons we also include in our metalanguage a *substitution* construct that allows one to insert data from the input directly into the template. The syntax of the substitution is `<: Expr :>`. Example of the substitution construct can be found, e.g., in Line 7 of Figure 11. This line is responsible for outputting the string "class " followed by the class name, that consists of the value of the variable *$state* and the word "State", i.e., the template evaluator first determines the value of the expression between `<:` and `:>` which must yield a string, and then replaces with this value the placeholder in the template. One can show that the substitution can be written as a combination of subtemplates and match-replaces [2] and, hence, it does not extend new functionality of the metalanguage. This combination of subtemplates and match-replaces is, however, very verbose and frequently used, so we decided to provide an explicit construct for the substitution construct.

## 8. Case Study: Reimplementation NunniFSMGen

In this section we show that the metalanguage introduced in Section 7 is indeed usable to specify a non-trivial code generator. As the case study we consider reimplementation of NunniFSMGen[1]. NunniFSMGen translates an abstract specification of a finite state machine into an implementation in Java, C or C++. It uses the state design pattern [13] to implement the state machine in the different output languages. The original implementation is a single stage code generator using print statements, while we reimplemented it using a parser, a model transformation and templates.

[1] http://sourceforge.net/projects/nunnifsmgen/ (accessed on May 5, 2011)

```
1  unparse[
2  <: match :>
3   <:Not($x)       =:> ~<:unparse($x):>
4   <:And($x1,$x2)=:> <:unparse($x1):> & <:unparse($x2):>
5   <:Or($x1,$x2)  =:> <:unparse($x1):> | <:unparse($x2):>
6   <:Br($x)        =:> ( <:unparse($x):> )
7   <:True          =:> true
8   <:False         =:> false
9  <: end :>
10 ]
```

**Figure 6.** Unparser for booleans expressed in template metalanguage.

| | | | |
|---|---|---|---|
| STANDBY | activate | WARMINGUP | warmup |
| STANDBY | deactivate | – | – |
| STANDBY | hotenough | ERROR | ! |
| STANDBY | maintenance | MAINTENANCE | maintenance |
| WARMINGUP | activate | – | – |
| WARMINGUP | deactivate | STANDBY | heateroff |
| WARMINGUP | hotenough | STANDBY | heateroff |
| WARMINGUP | maintenance | MAINTENANCE | heateroff |
| ERROR | activate | – | – |
| ERROR | deactivate | – | – |
| ERROR | hotenough | – | – |
| ERROR | maintenance | MAINTENANCE | – |
| MAINTENANCE | activate | STANDBY | initialize |
| MAINTENANCE | deactivate | – | – |
| MAINTENANCE | hotenough | – | – |
| MAINTENANCE | maintenance | – | – |

**Figure 7.** Transition table for a central heating system.

### 8.1 Functionality of NunniFSMGen

NunniFSMGen is an open source tool developed by the Swiss-based company NunniSoft. NunniFSMGen can translate an abstraction specification of a state machine, given as a transition table, to a corresponding implementation in Java, C or C++. The transition table is a set of transition rules of the form `startState event nextState action`: if a state machine residing in the `startState`, receives an `event`, then the `action` is executed and the state of the machine is changed to `nextState`.

In general, the `action` is implemented as a method invocation. If, however, `-` is specified as an `action`, then no action is required when the transition is executed. Similarly, if `-` is specified as a `nextState`, then a transition rule does not cause a change of state. Finally, the `action` can also contain an exclamation mark,`!`. The exclamation mark defines that the action must throw an exception and after that the state machine will go the `errorState`.

**Example 8.1.** Figure 7 shows the transition table of a central heating system example delivered with NunniFSMGen. Figure 8 presents a snippet of a parser used to obtain an abstract syntax tree representation of the state machine. The model transformation presented in Section 8.2 uses this abstract syntax tree as input.

### 8.2 State Machine Implementation

NunniFSMGen implements a variant of the state pattern using the transition table as input, where the events are handles and the states are implemented as concrete states. The transition table cannot be used directly for generating the code using our template metalanguage. The state pattern has a hierarchical structure, where each state implements handlers for each event, while the transition table is list of vectors pointing from the `startState` to the `nextState`.

```
rules([
 transition("STANDBY","activate",
    nextstate("WARMINGUP"),action("warmup")),
 ...
 transition("MAINTENANCE","maintenance",
                          nonextstate,noaction)
])
```

**Figure 8.** Part of the abstract syntax tree of the transition table of the central heating system.

```
afsm(
 transitions([
  transition( "STANDBY", events([
    event("activate",
          nextstate("WARMINGUP"), action("warmup") ),
    event("deactivate", nonextstate, noaction ),
    event("hotenough", nextstate("ERROR"),
                                        erroraction ),
    event("maintenance", nextstate("MAINTENANCE"),
                               action("maintain") ) ])),
    ...
 ]),
 events([ "activate", "deactivate",
          "hotenough", "maintenance" ]),
 actions([ "warmup","maintain","heateroff",
                                      "initialize" ])
)
```

**Figure 9.** Abstract implementation of the state design pattern of the heater transition table.

A model transformation is necessary to map the vector based transition table to the hierarchical based state design pattern. First, the transformation collects all unique events from the transition rules and stores them in the set `events`. Then, it collects all unique actions from the transition rules and stores them in the set `actions`. Finally, it creates a new `transition` rule for each state and adds for each event a triple with the `event`, the `nextState` and the `action`. Transition rules are collected in the set `transitions`.

Implementation of this model transformation is straight-forward. It contains seven equations based on 18 sub-equations. The abstract implementation of the state design pattern of the central heating system is shown in Figure 9. The square brackets denote list, an additional feature of the used tree format.

### 8.3 Original Code Generator

The original implementation of NunniFSMGen is a generator based on print statements written in Java. Its main class contains a simple parser for the input transition table, constructing a one-to-one in memory representation of the transition table.

For each output language a code generator class is implemented, containing the generator logic and object code fragments. Based on the selected output language a code generator class is instantiated and provided with the loaded input data.

We consider such a code generator class as a single-stage code generator, since the different code generator classes share a significant amount of code. The shared code is not factorized out in a model transformation. Furthermore, since it is a single-stage code generator, the model transformations are mostly entangled between object code artifacts. During the initialization of the code generator, only the set of events and the set of states are calculated.

To illustrate the adverse effects of the single-stage code generation used in NunniFSMGen consider the generation of the particular code for an event. Implementation of the required behavior is selected based on a set of conditions distinguishing

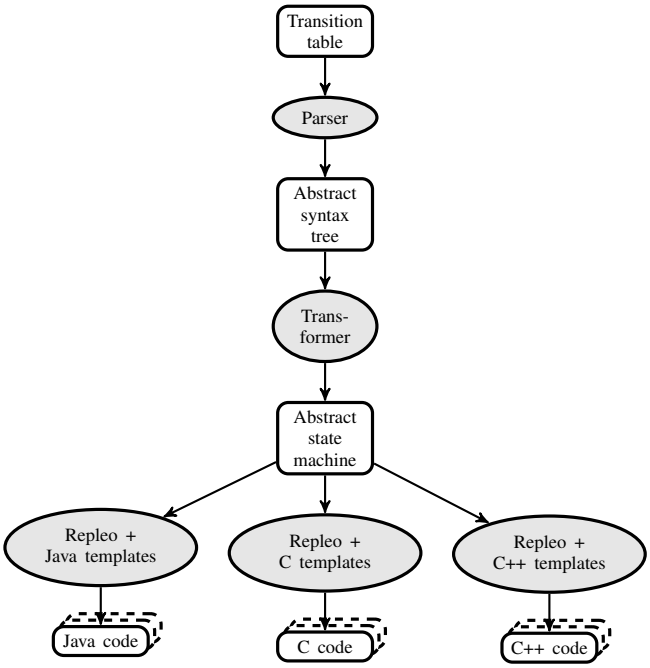


**Figure 10.** Architecture of the reimplemented NunniFSMGen.

between the following five kinds of events: no state change, no action (`startState event - -`); no state change, with action (`startState event - action`); state change, no action (`startState event nextState -`); state change, with action (`startState event nextState action`); state change to `errorState` with error action (`startState event errorState !`). Since NunniFSMGen exists of three almost independent single-stage code emitters for C, C++ and Java, the set of conditions corresponding to the choice of the implementation is cloned between the code emitters. Moreover, similarity between C++ and Java means that the code emitters for these languages are almost identical, except for the object code. In case of the code emitter for C, the meta code is almost the same, but the approach used to implement the exception handling in the object code differs from the C++ and Java implementations.

### 8.4 Reimplemented Code Generator

The reimplementation of NunniFSMGen is based on a two-stage architecture using a parser, a model transformation phase and templates. The previous discussed input model parser and model transformations are output language independent. The output is an abstract syntax of a state machine, which is still output language independent. The templates in the second stage contain the additional information in the object code to implement an output language specific instantiation of a state machine. In this case for output languages, C, C++ and Java, a set of templates is defined.

Code shared by the templates is limited to the meta code responsible for traversing the input data tree: while all templates use the same abstract representation of the state machine as input data, tailored model transformations are unnecessary for each one of the output languages. In this way the model transformation has been separated from the part of the code generator that depends on the output language. The architecture of the reimplemented NunniFSMGen is shown in Figure 10. The templates are evaluated by Repleo [2]. Repleo is a template engine based on the unparser-complete metalanguage as defined in this article.

```
1  transitions[
2  <: match :>
3    <:[transition( $state, $events),
4          $transitions] =:>
5       template[
6          <: $state + "State.java" :>,
7          class <: $state + "State" :> extends State
8          {
9          ...
10         }
11      ]
12      <: transitions($transitions) :>
13   <: [] =:>
14 <: end :>
15 ]
```

**Figure 11.** Java snippet of the template implementation.

```
1  eventcode[
2  static int <: $state + "State" + $event :>
3         ( struct FSM *fsm,
4           void * o ) {
5    int ret = 0;
6    ...
7    <: match $nextstate :>
8      <: nextstate($nextstatename) =:>
9        if ( ret < 0 )
10           fsm->changeState( fsm,
11              &m_ErrorState );
12       else
13           fsm->changeState( fsm,
14              &<: "m_" + $nextstatename
15                    + "State" :> );
16     <: nonextstate =:>
17       if ( ret < 0 )
18           fsm->changeState( fsm,
19              &m_ErrorState );
20   <: end :>
21   ...
22   return ret;
23 }
24 ]
```

**Figure 12.** C snippet of the template implementation.

Figures 11 and 12 show snippets of the reimplementation using templates in Java and C, respectively. For the sake of brevity we do not include the reimplementation in C++, which can be found in [2]. Observe the use of the built-in template *template* (e.g., Figure 11, Lines 5–11). The template engine evaluates the second argument of *template* and stores the result in a file with the name obtained by evaluating its first argument. So, if *$state* is STANDBY then, evaluation of Lines 5–11 of Figure 11 results in storing the Java code generated by Lines 7–10 in the file called STANDBYState.java.

### 8.5 Evaluation

The introduction of templates based on an unparser-complete metalanguage has improved the NunniFSMGen implementation. The new implementation uses a two-stage architecture, where the old implementation almost directly generates code from the transition table in a single-stage architecture. The two-stage architecture including the unparser-complete metalanguage enforced a clear separation between the model transformation and the code generation. The first stage parses and rewrites the transition table in order to get an abstract implementation of the state design pattern. The second stage is responsible for generating the concrete code for the different output languages.

We expect that the strict separation between the model transformation and the code generation reduces the implementation effort required to add a new output language. Furthermore, the original code generator contains code clones between emitters for the different output languages. Presence of clones might lead to diverging evolution of the emitters, i.e., a risk of inconsistent behavior of, e.g., Java and C implementations of the same state machine. By separating model transformation encapsulating the shared code from code generation depending on the target language, we successfully eliminate the cross-emitter clones. Clone elimination is also apparent in the counts of source lines of code in the original implementation vs. the reimplementation: 1430 vs. 738 source lines of code.

Clearly, while the templates foster reuse and help in eliminating of the cross-emitter clones, there is still the potential for cloned templates. No cloned templates were found in the reimplementation. In general, cloned templates are an indication of a non-optimal design.

## 9. Related work

Numerous template engines and associated metalanguages can be found both in the academia and the industry [4, 16, 21, 25]. Unfortunately, formal expressivity requirements of such languages have attracted less attention from the research community with [21] being the only notable exception we are aware of. Similarly to [21] we advocate the separation between the view and the model.

ERb [16] is a text template interpreter for the programming language Ruby. ERb introduces special syntax constructs to embed Ruby code in a text file. The metalanguage of ERb is Ruby and thus Turing-complete, as a result there is no restriction on the code ERb can generate. However, to implement an unparser, a developer should take into account that metavariables are globally accessible and writable. Hence, it is necessary to specify a stack mechanism in the meta-code explicitly.

JSP [4] is a template based system developed by Sun Microsystems. It is designed for generating dynamic web pages and XML messages in Java-based enterprise systems, where the evaluation is tuned for performance. The Java language is available as metalanguage in JSP pages, although special tag libraries are available to provide concise constructs. JSP supports variables and a `for` loop, and hence, JSP goes beyond the unparser-completeness. Moreover, it has the same problem with scoping of meta-variables as ERb. It is necessary to use different scopes for different variables, introducing unwanted boilerplate code.

Velocity [25] is a template evaluator for Java. It provides a basic template metalanguage, called Velocity Template Language, to reference Java objects. The metalanguage of Velocity is also Turing-complete, as it is possible to set variables. In case of implementing unparsers, Velocity has the same problem with the variable scopes as ERb and JSP. As a consequence temporary variables are necessary when a subtemplate is recursively called.

The main lessons learned from the related template engines is that not only the available constructs are important, but also the variable scope handling in the evaluator. Furthermore, the metalanguage should be powerful enough to handle all (path-closed) regular tree languages. For example, StringTemplate [21] is not capable of referring to the different children of a node if the children have the same label. An extra input data transformation is necessary to guarantee the input tree can be accepted by StringTemplate.

We have used the ideas of unparser-completeness to construct the new version of a metalanguage for our template engine Repleo [3]. Repleo supports syntax-safe template evaluation, i.e., it guarantees that the generator always instantiates a sentence belonging to the desired output language or in case it cannot produce such a sentence, that the evaluator terminates with an error message. The first version of Repleo [3] was based on a metalanguage in-

spired by ERb [16] and the ideas of [21]. Two major differences can be found between the first version of the metalanguage and the current one. First, the metalanguage of [3] provides a broader set of expressions, allowing the template designer to violate the separation of concerns principle, e.g., by defining mathematical operations on the input data. Second, the metalanguage of [3] did not support recursive template evaluation, which is necessary for unparser-completeness. The reimplementation of Repleo [2] is based on the notion of unparser-completeness, such as presented in this article.

Finally, *unparse* and *desugar* $\circ$ *parse* can be seen as inverse to each other. Such properties as linearity (cf. Theorem 4.6) are often assumed in studies of program inversion [14]. Furthermore, relation between parsers and unparsers (pretty printers) was studied in [1, 23]. We stress, however, that our goal is different: unlike [1, 23] we do not focus on finding a unified framework for parsers and unparsers but apply these interrelated notions to formalize the requirements for a metalanguage of a template engine in the context of the model view controller architecture.

## 10. Conclusions

In this paper, following the ideas of Parr [21], we have presented a formal notion of unparser-completeness. Unparser-completeness of a metalanguage provides the balance between expressivity and restrictiveness. On one hand, the metalanguage is expressive enough to implement an unparser, and, hence, can instantiate any semantically correct program in the object language. On the other hand, the metalanguage is restricted enough to enforce the model-view separation in terms of Parr [21].

Next, we showed that a linear deterministic top-down tree-to-string transducer is powerful enough to implement an unparser. Using the notion of the top-down tree-to-string transducer we have shown that unparser-completeness is a weaker notion than Turing completeness, i.e., unparser-complete meta-languages are not necessarily Turing-complete. The enforcement of separation of concerns is also met, as this transducer does not allow to express calculations or modify the model.

At the end, we showed that unparser-completeness is not only a theoretical framework. We reimplemented NunniFSMGen using templates based on an unparser-complete metalanguage. This metalanguage enforces us to use a two-stage architecture for the reimplementation, which resulted in an improved separation between model transformations and code emitting.

## References


[1] A. Alimarine, S. Smetsers, A. van Weelden, M. C. J. D. van Eekelen, and M. J. Plasmeijer. There and back again: arrows for invertible programming. In *ACM SIGPLAN workshop on Haskell*, pages 86–97, New York, NY, USA, 2005. ACM.

[2] B. J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. PhD thesis, Technische Universiteit Eindhoven, 2010.

[3] B. J. Arnoldus, J. W. Bijpost, and M. G. J. van den Brand. Repleo: a Syntax-Safe Template Engine. In *GPCE '07*, pages 25–32, New York, NY, USA, 2007. ACM Press.

[4] H. Bergsten. *Javaserver Pages*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2002.

[5] M. G. J. van den Brand, P. E. Moreau, and J. J. Vinju. A generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings Software*, 152(2):70–78, 2005.

[6] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, 1996.

[7] S. Burbeck. Applications Programming in Smalltalk-80: How to use Model-View- Controller (MVC), 1992.

[8] L. G. W. A. Cleophas. Private communication, September 2009.

[9] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: http://www.grappa.univ-lille3.fr/tata (accessed on November 30, 2010), 2008. release November, 18th 2008.

[10] F. L. Deremer. *Practical Translators for LR(k) languages*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1969.

[11] J. Engelfriet. Tree Automata and Tree Grammars. Manual written lecture notes, 1974.

[12] J. Engelfriet, G. Rozenberg, and G. Slutzki. Tree transducers, L systems, and two–way machines. *Journal of Computer and System Sciences*, 20(2):150–202, 1980.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1995.

[14] R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori, editor, *Programming Languages and Systems*, volume 2895 of *LNCS*, pages 246–264. Springer, 2003.

[15] J. Hartmanis. Context-free languages and Turing machine computations. In *Symposia in Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 42–51. Amer Mathematical Society, 1967.

[16] J. Herrington. *Code Generation in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[17] S. C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical Report 32, Bell Laboratories, Murray Hill, NJ, USA, 1975.

[18] P. Klint, T van der Storm, and J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM '09*, pages 168–177, Los Alamitos, CA, USA, 2009. IEEE Computer Society Press.

[19] G. E. Krasner and S. T. Pope. A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[20] D. C. Oppen. Pretty printing. Technical Report STAN-CS-79-770, Computer Science Department, Stanford University, October 1979.

[21] T. J. Parr. Enforcing Strict Model-View Separation in Template Engines. In *WWW '04: International Conference on World Wide Web*, pages 224–233, New York, NY, USA, 2004. ACM Press.

[22] N. Ramsey. Unparsing expressions with prefix and postfix operators. *Software: Practice & Experience*, 28(12):1327–1356, 1998.

[23] T. Rendel and K. Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. In *ACM SIGPLAN Haskell symposium*, pages 1–12, New York, NY, USA, 2010. ACM.

[24] T. Sheard. Accomplishments and Research Challenges in Metaprogramming. In *SAIG*, volume 2196 of *LNCS*, pages 2–44, London, UK, 2001. Springer.

[25] T. Sturm, J. von Voss, and M. Boger. Generating Code from UML with Velocity Templates. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *UML*, volume 2460 of *LNCS*, pages 150–161. Springer, 2002.

[26] M. G. J. van den Brand, J. S. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *LNCS*, pages 21–44. Springer, 2002.

[27] J. Virágh. Deterministic ascending tree automata I. *Acta Cybernetica*, 5:33–42, 1981.

[28] E. Visser. Stratego: A language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In *RTA '01*, volume 2051 of *LNCS*, pages 357–361, Berlin, Heidelberg, 2001. Springer.

[29] D. S. Wile. Abstract syntax from concrete syntax. In *ICSE*, pages 472–480, New York, NY, USA, 1997. ACM Press.